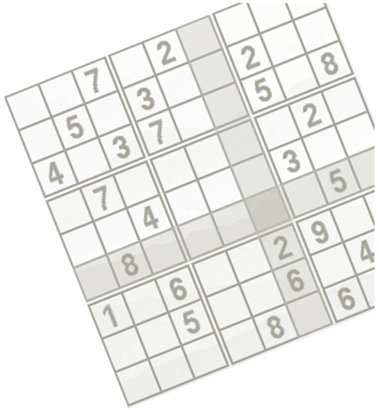


Constraint Programming

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Local search algorithms

Constraint Satisfaction Problem (CSP) consists of:

- a finite set of **variables**
- **domains** – finite sets of possible values for variables
- a finite set of **constraints**
  - constraint **arity** = the number of constrained variables

A feasible solution of a constraint satisfaction problem is a complete consistent assignment of values to variables.

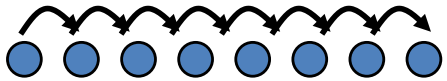
- **complete** = each variable has assigned a value
- **consistent** = all constraints are satisfied

Looking for a solution

The goal: **find a complete and consistent instantiation of variables**

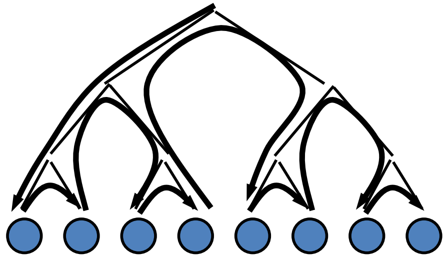
Two **core solving approaches**:

– **exploring complete but possibly inconsistent assignments** until a consistent assignment is found



  - generate and test, local search

– **extending a partial consistent assignment** until a complete assignment is reached



  - backtracking and its extensions

We can explore assignments in two ways:

– **systematically** (explore all possible assignments systematically)

  - a complete method, but could be too slow

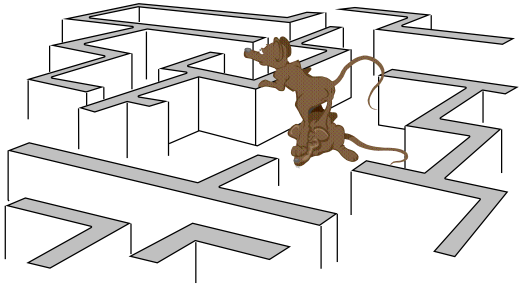
– **non-systematically** (some assignments can be skipped)

  - an incomplete method, but can found solution much faster

Note:

We will use constraints in a *passive way*, just to verify whether the given assignment (even partial) satisfies the constraint.

Search techniques



Work plan:

- start simple (with a trivial algorithm)
- find weaknesses of the algorithm
- repair the weaknesses to get better algorithms

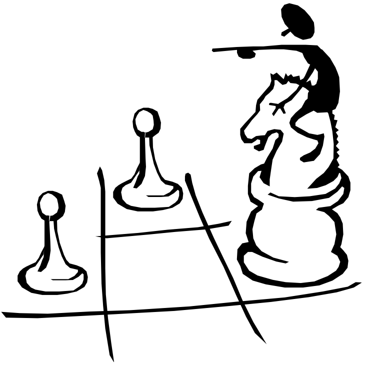
In particular:

- start with **generate and test** method
- **improve the generator**
  - local search methods (HC, RW, TS, GSAT, GENET, SA)
- **merge the generator with the tester**
  - backtracking methods
  - improvements of chronological backtracking
    - backjumping, dynamic backtracking, backmarking

Generate and test (GT)

Probably the most general problem solving method

1) generate a candidate for solution
2) test if the candidate is really a solution

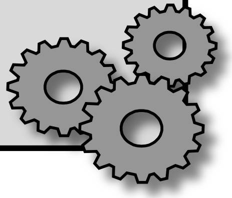


How to apply GT to CSP?

1) assign values to all variables
2) test whether all the constraints are satisfied

GT **explores complete but inconsistent assignments** until a (complete) consistent assignment is found.

```
procedure GT(X:variables, C:constraints)
    V ← construct a first complete assignment of X
    while V does not satisfy all the constraints C do
        V ← construct systematically a complete assignment next to V
    end while
    return V
```



Weaknesses and improvements of GT

The greatest weakness of GT is **exploring too many “visibly” wrong assignments**.

Example:

X::{1,2}, Y::{1,2}, Z::{1,2}   X = Y, X $\neq$ Z, Y > Z

X	1	1	1	1	2	2	2
Y	1	1	2	2	1	1	2
Z	1	2	1	2	1	2	1

How to improve GT?

- **smart generator**
  - the next assignment improves over the current assignment
  - the core idea of local search techniques
- **merged generate and test stage** (earlier detection of clash)
  - constraints are tested as soon as all involved variables are instantiated
  - backtracking

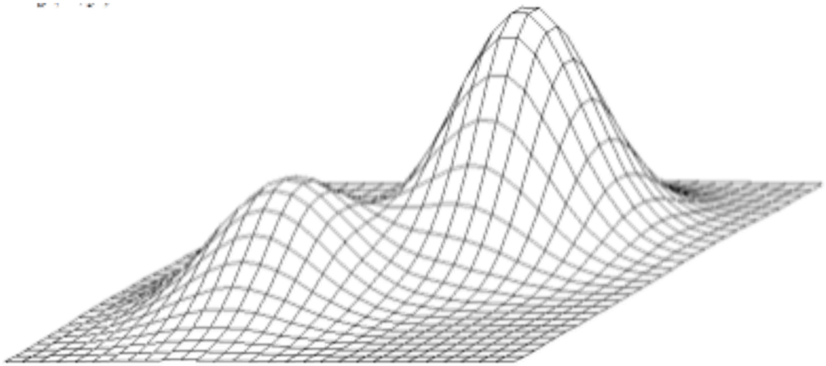
Local search

Generate and test explores complete but inconsistent assignments until a complete consistent assignment is found.

Weakness of GT – the generator does not exploit fully the result of testing

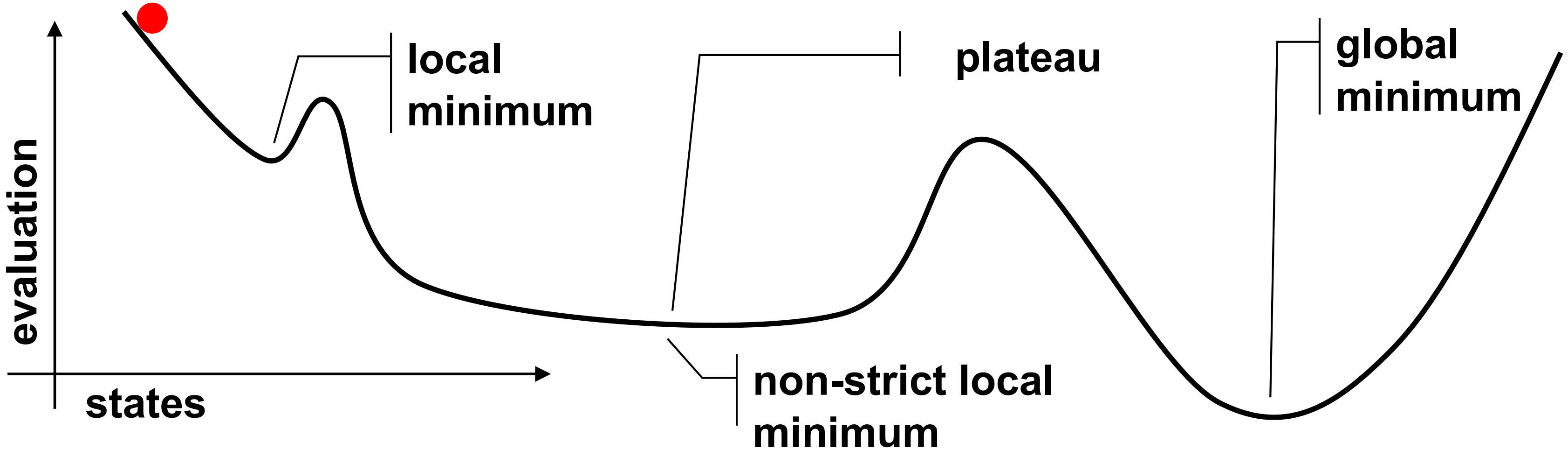
The next assignment can be constructed in such a way that constraint violation is smaller.

- only “small” (**local**) **changes** of the assignment are allowed
- the next assignment should be “better” than the current one
  - better = more constraints are satisfied
- assignments are not necessarily generated systematically
  - we lost completeness, but
  - we (hopefully) get better efficiency



Local search - Terminology

- **state** - a complete assignment of values to variables
- **evaluation** - a value of the objective function (# violated constraints)
- **neighbourhood** - a set of states locally different from the current state (the states differ from the current state in the value of one variable)
- **local optimum** - a state that is not optimal and there is no state with better evaluation in its neighbourhood
- **strict local optimum** - a state that is not optimal and there are only states with worse evaluation in its neighbourhood
- **non-strict local optimum** - local optimum that is not strict
- **plateau** - a set of neighbouring states with the same evaluation
- **global optimum** - the state with the best evaluation



Hill Climbing

Hill climbing is perhaps the most known technique of local search.

- start at randomly generated state
- look for the best state in the neighbourhood of the current state
  - **neighbourhood** = differs in the value of any variable
  - neighbourhood size = $\Sigma_{i=1..n}(D_i-1)$ (= $n*(d-1)$)
- "escape" from the local optimum via restart

Algorithm Hill Climbing

```
procedure hill-climbing(Max_Steps)
    restart: s ← random assignment of variables;
    for j:=1 to Max_Steps do                % restricted number of steps
        if eval(s)=0 then return s
        if s is a strict local minimum then
            go to restart
        else
            s ← neighbourhood with the smallest evaluation value
        end if
    end for
    go to restart
end hill-climbing
```

Minton, Johnston, Laird (1997)

Min-Conflicts

Observation:

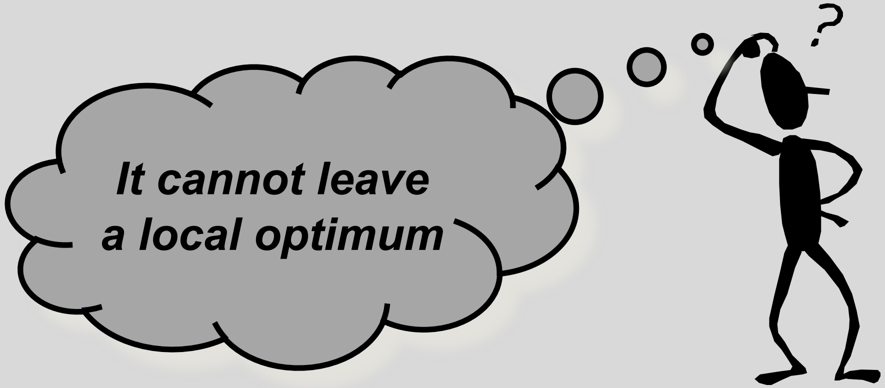
- the hill climbing neighbourhood is pretty large (n*(d-1))
- only change of a conflicting variable may improve the evaluation

Min-conflicts method

- select randomly a variable in conflict and try to improve it
  - **neighbourhood** = different values for the selected variable *i*
  - neighbourhood size = (D_i-1) (= (d-1))

Algorithm Min-Conflicts

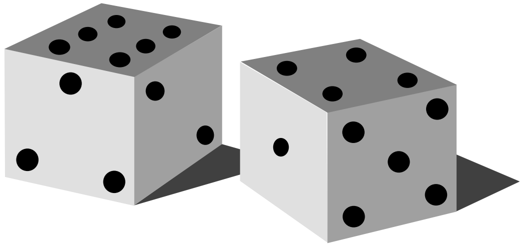
```
procedure MC(Max_Moves)
    s ← random assignment of variables
    nb_moves ← 0
    while eval(s)>0 and nb_moves<Max_Moves do
        choose randomly a variable V in conflict
        choose a value v' that minimises the number of conflicts for V
        if v' ≠ current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end MC
```



Random Walk

How to leave a local optimum without restarting (i.e. via a local step)?

- By adding some “noise” to the algorithm!



Random walk

- **a state from the neighbourhood is selected randomly** (e.g., the value is chosen randomly)
- such technique can hardly find a solution
- so it needs some guide
  - Random walk can be combined with the heuristic guiding the search process via probability distribution:
    - p - probability of using a random step
    - (1-p) - probability of using the heuristic guide

Min-Conflicts Random Walk

MC guides the search (i.e. satisfaction of all the constraints) and RW allows us to leave the local optima.

Algorithm Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
    s $\leftarrow$ random assignment of variables
    nb_moves $\leftarrow$ 0
    while eval(s)>0 and nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose randomly a variable V in conflict
            choose a value v' that minimises the number of conflicts for V
        end if
        if v' $\neq$ current value of V then
            assign v' to V
            nb_moves $\leftarrow$ nb_moves+1
        end if
    end while
    return s
end MCRW
```

$0.02 \leq p \leq 0.1$

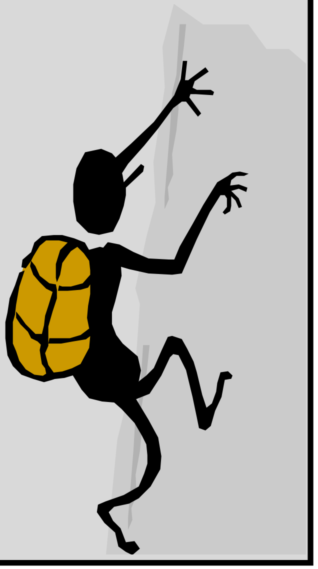
Steepest Descent Random Walk

Random walk can be combined with the hill climbing heuristic too.

Then, no restart is necessary.

Algorithm Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
    s ← random assignment of variables
    nb_moves ← 0
    while eval(s)>0 and nb_moves<Max_Moves do
        if probability p verified then
            choose randomly a variable V in conflict
            choose randomly a value v' for V
        else
            choose a move <V,v'> with the best performance
        end if
        if v' ≠  current value of V then
            assign v' to V
            nb_moves ← nb_moves+1
        end if
    end while
    return s
end SDRW
```



Tabu search

Observation:

Being trapped in a local optimum is a special case of cycling.

How to avoid cycles in general?

- **remember already visited states** and do not visit them again
  - memory consuming (too many states)
- it is possible to remember just a few last states
  - prevents „short“ cycles

- **Tabu list** = a list of forbidden states
  - a state can be represented by a selected attribute
    - ⟨variable, value⟩ - describes the change of a state (the previous value)
  - the tabu list has a fix length k (tabu tenure)
    - „old“ states are removed from the list when a new state is added
  - a state included in the tabu list is forbidden (it is tabu)
- **Aspiration criterion** = re-enabling states that are tabu
  - i.e., it is possible to visit a state even if the state is tabu
  - **example:** the state is better than any state visited so far

Tabu search

The tabu list prevents short cycles.

It allows only the moves out of the tabu list or the moves satisfying the aspiration criterion.

Algorithm Tabu Search

```
procedure tabu-search(Max_Iter)
    s ← random assignment of variables
    nb_iter ← 0
    initialise randomly the tabu list
    while eval(s)>0 and nb_iter<Max_Iter do
        choose a move <V,v'> with the best performance among the non-tabu
            moves and the moves satisfying the aspiration criteria
        introduce <V,v> in the tabu list, where v is the current value of V
        remove the oldest move from the tabu list
        assign v' to V
        nb_iter ←  nb_iter+1
    end while
    return s
end tabu-search
```

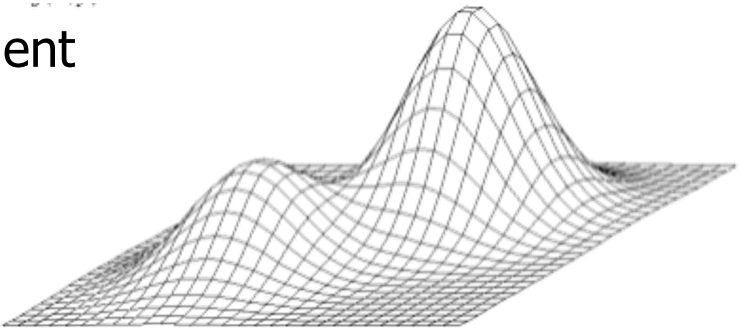
Local Search at Glance

LS methods explore complete but possible inconsistent assignments until a consistent assigned is found

  - opposite to GT, they generate a new assignment based on the current assignment with the goal to increase the number of satisfied constraints

Local search algorithm is defined by:

- **neighbourhood** of the current assignment (state) and a method to **select the next assignment** from the neighbourhood (**intensification**)
  - HC heuristic – select the best assignment different at one variable from the current assignment
    - sometimes, the first better assignment from the neighbourhood is taken
  - MC heuristic – select the best assignment different at one selected conflict variable from the current assignment
- a method for **escaping from a local optimum** (**diversification**)
  - restart – start in a completely new assignment
  - RW – select the next assignment randomly
  - Tabu – forbid some assignments



Local Search for SAT

Many problems can be formulated as problems of Boolean SATisfiability = **satisfying a logical formula** in a conjunctive normal form (CNF)

- **CNF** = conjunction of clauses
- **clause** = disjunction of literals (constraint)
- **literal** = atomic variable or its negation

Example:

$(A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee \neg A)$

- Similarly to a CSP, SAT is also an NP-complete problem so no fast (polynomial) solving algorithm can be expected.
- Local search can find a solution to pretty large formulas.

Notes:

- satisfaction formula in a disjunctive normal form can be decided fast
- SAT is a special case of a CSP and vice-versa, any CSP can be translated to a SAT problem

Algorithm GSAT

The GSAT method solves SAT problems by flipping the values of variables.

The goal is to maximize the (weighted) number of satisfied clauses.

Algorithm GSAT

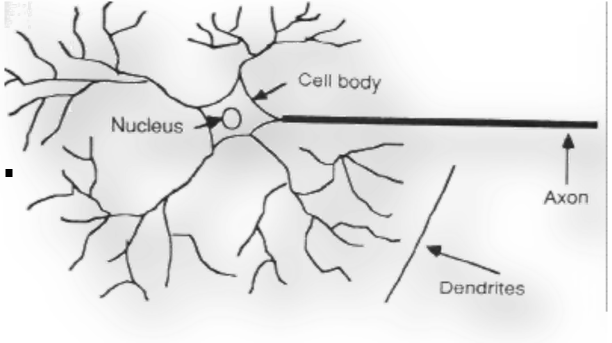
```
procedure GSAT(A,Max_Tries,Max_Moves)
    A: is a CNF formula
    for i:=1 to Max_Tries do
        S ←  random assignment of variables
        for j:=1 to Max_Moves do
            if A satisfiable by S then return S
            V ←  the variable whose flip yield the most important raise
                 in the number of satisfied clauses
            S ← S with V flipped
        end for
    end for
    return the best assignment found
end GSAT
```

GSAT and heuristics

GSAT can be combined with various heuristics improving its practical performance (especially for so called structured problems) :

- **Random-Walk**
  - can be used exactly as in MCRW
- **Clause weights**
  - some clauses remain unsatisfied even after several iterations of the inner loop of GSAT ➔ different clauses have different importance in formula satisfaction
  - satisfaction of “hard” clauses can be preferred by increasing their weights in the clause selection process
  - the algorithm can learn the weights itself
    - all clauses have identical weight at the beginning
    - after each iteration, the weights of unsatisfied clauses are increased
- **Solution averages**
  - in the GSAT algorithm each iteration starts from a random assignment of variables – hence the last reached assignment is “forgotten”
  - we can reuse the common parts of found assignments
    - the new assignment after restart is taken from the last assignments of previous two iterations by keeping the same parts and setting the remaining variables randomly

Connectionistic approach



- Based on idea of representing the problem as a **network of connected simple processors**.
    - processors have several states (usually only two – on/off).
    - The next state of the processor is derived from the states of connected processors (the connection strengths may be different).
- The goal is to find a stable state of the network, i.e., the processors are no more changing their states.
- This stable state represents a solution to the problem.

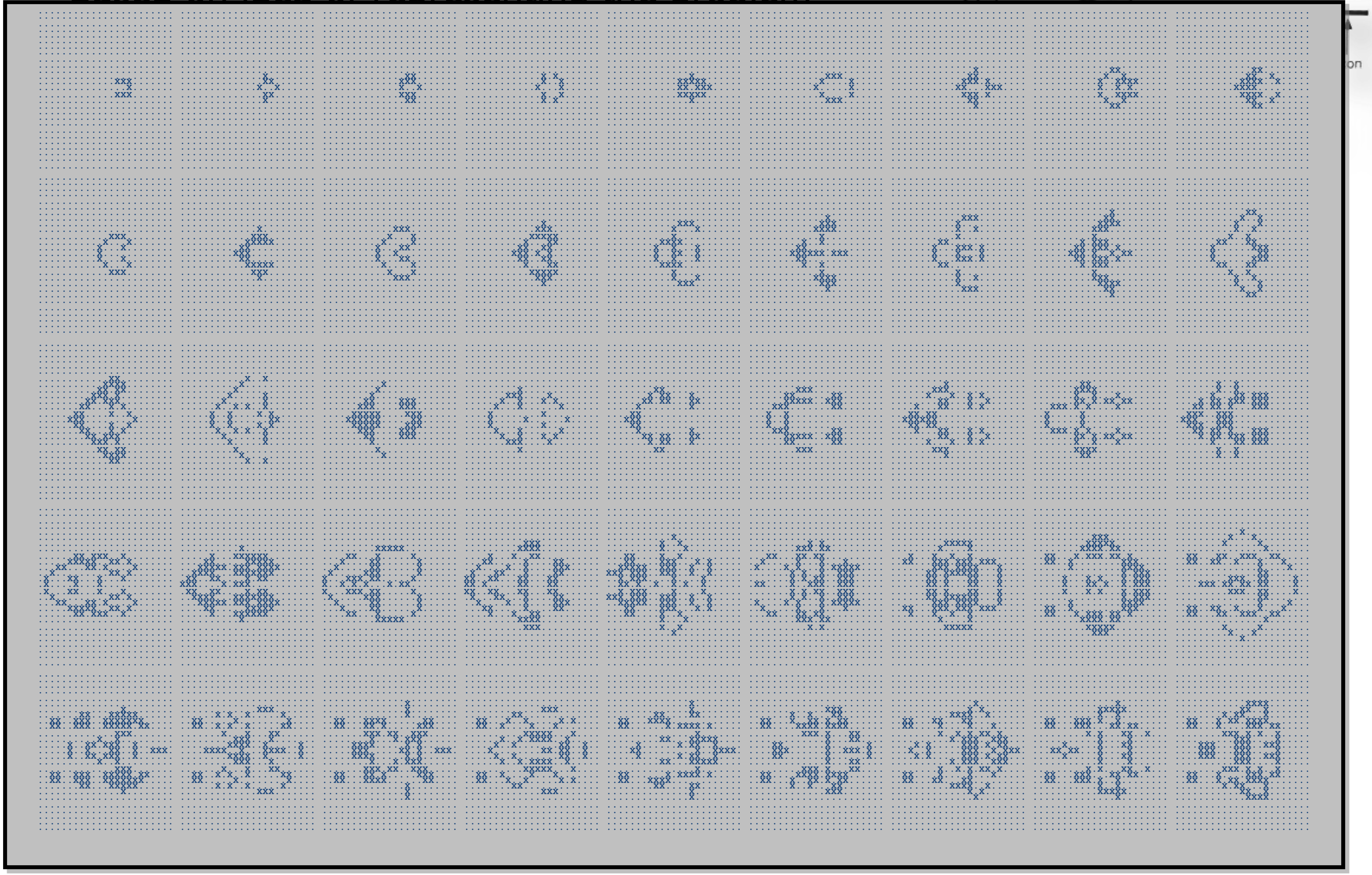
Features:

- massive parallelism (problems can be solved faster)
- blackbox (not clear what is happening inside)

- Probably the most known representative is an artificial **neural network** (NN)
- A similar principle is used in **celular automata.**

Connectionistic approach

- Based on idea of representing the problem

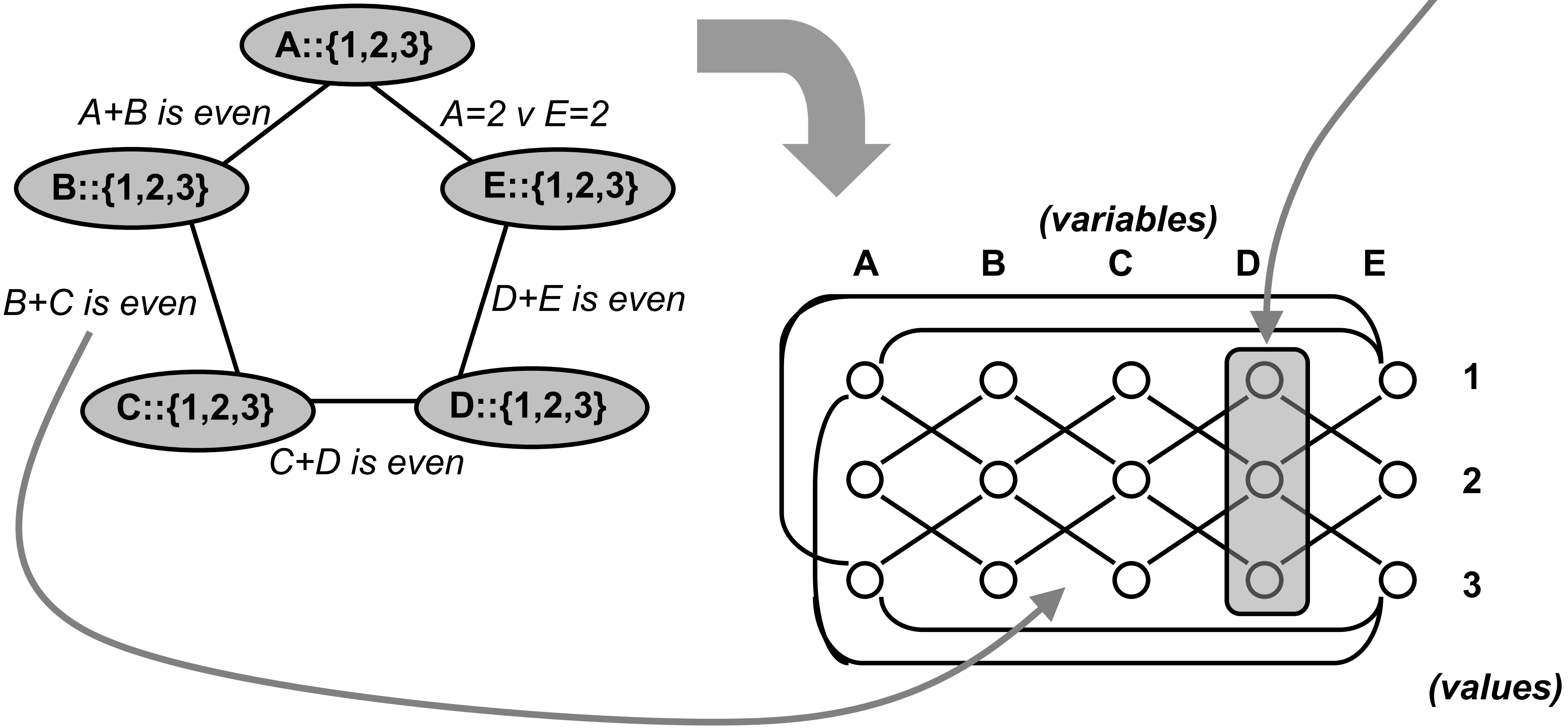


GENET – Binary CSP as an ANN

Each variable is modelled as a cluster of "neurons" (each value models a single neuron)

Two neurons are connected by the inhibition link with negative weight if the corresponding values are incompatible.

Example:



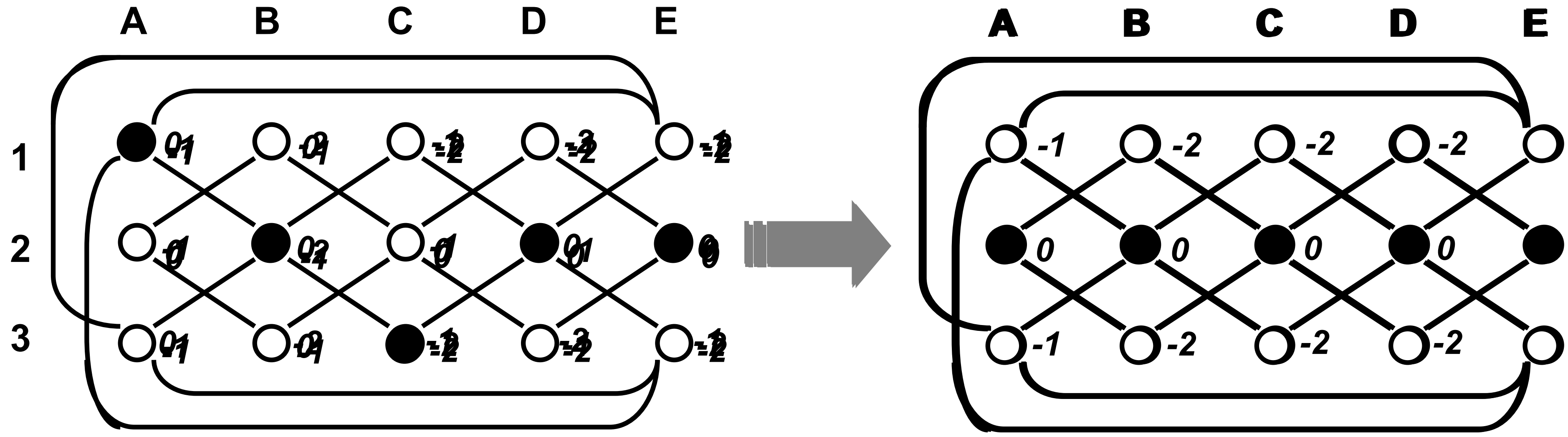
GENET computation

At the beginning, one active neuron is selected in each cluster.

Neurons change states in a **synchronous way** (all together)

- based on the inputs (Σ w*s – weighed sum of states of connected neurons)
- For each cluster, the neuron with the largest input is activated

The computation stops in a **stable state**.

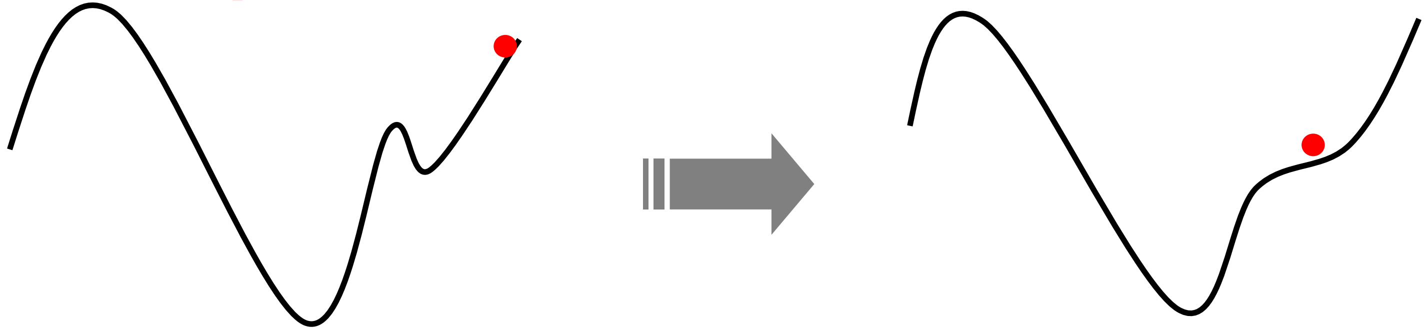


● = „active“ neuron; the numbers indicate inputs to neurons

Escape from local optimum

What if we reach a stable state that is not a solution?

- So far we used either restart or “noise”.
- We can try to modify the **space of state evaluations**.
  - How? By modifying the evaluation function!
    - **dynamic local search**

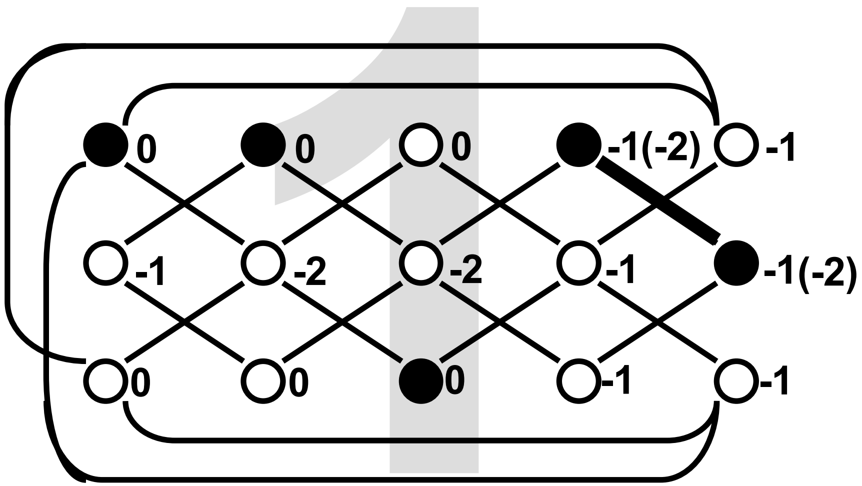


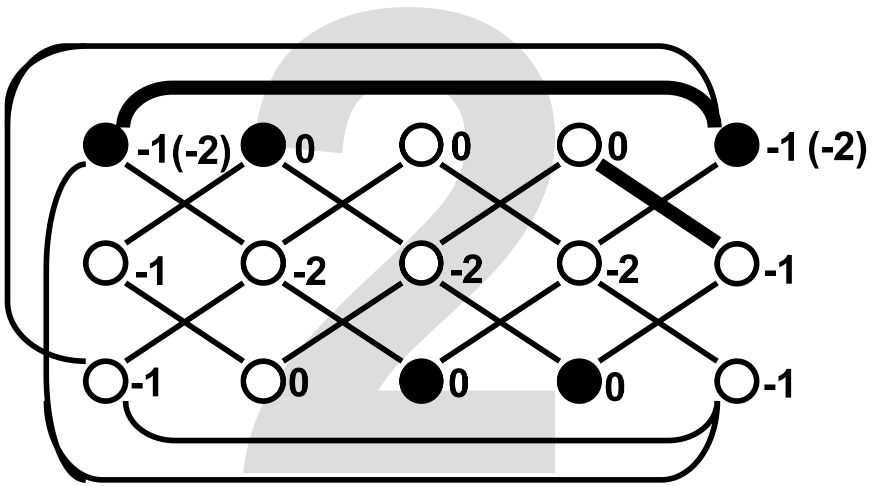
This can be done by modifying the **weight of connections** in GENET!

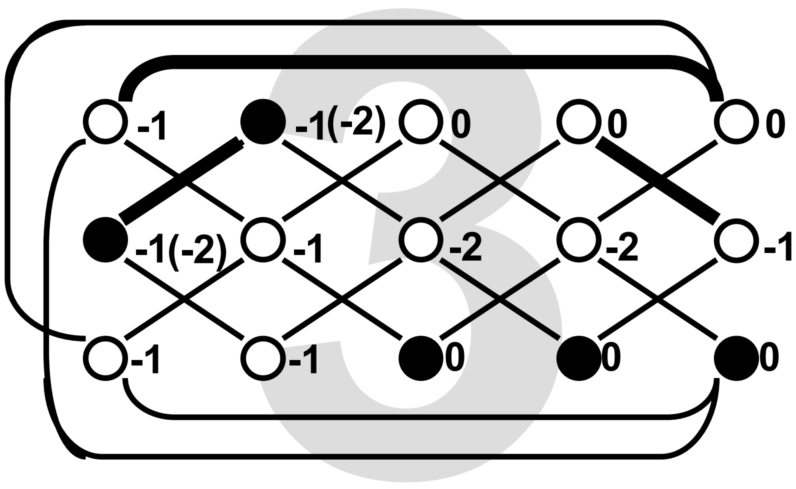
- If there is a connection between two active neurons (= constraint violation), increase the weight of the connection.
  - $new\_weight_{,y} = old\_weight_{x,y} – s_x * s_y$
- This also changes the evaluation function (**Guided Local Search**).

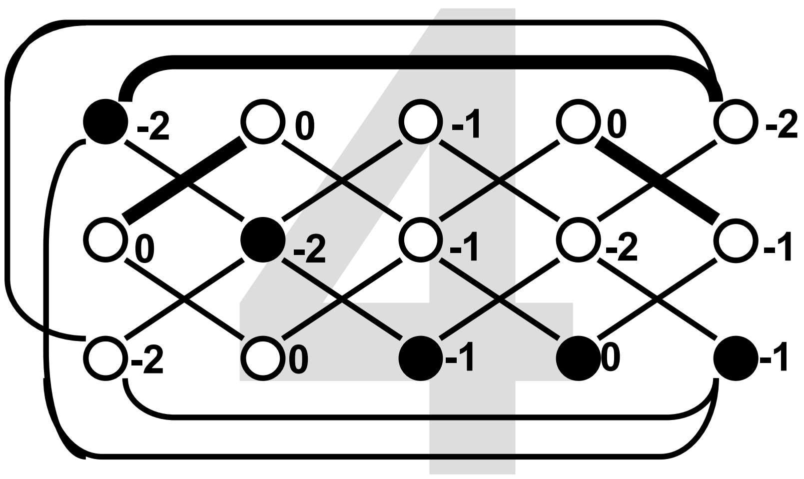
Example of changing connection weights

In local optimum we **strengthen weights** of violated connections (which makes the state instable).





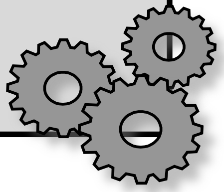




...

Algorithm GENET

```
procedure GENET(connectionist network)
    one arbitrary node per cluster is switched on;
    repeat
        repeat  % network convergence
            modified ← false;
            for each cluster C do in parallel
                on_node ←  currently switched on node in cluster C;
                label_set ← the set of nodes in C which input are maximum;
                if on_node is not in label_set then
                    switch off on_node;
                    modified ← true;
                    switch on arbitrary node in label_set;
                end if
            end for
        until not modified
        if sum of input to all switched-on nodes < 0 then
            for each connection c connecting nodes x & y do in parallel
                if both x and y are switched on then
                    decrease the weight of c by 1;
            end for
        end if
    until input to all switched-on nodes are 0
end GENET
```



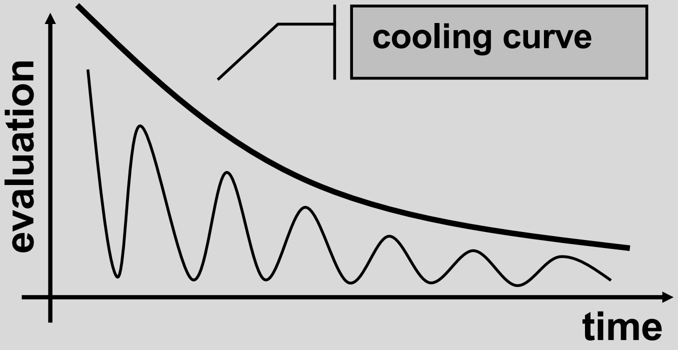
Based on the idea of **simulating the process of metal cooling**.

- Higher temperature means faster movement of atoms so the probability of changing position is higher.
- By cooling down, the atoms "try" to find the "best" position – the position with the smallest energy.

A very similar process can be modelled in optimisation algorithms:

- so called **simulated annealing**:
  - start with a random state
  - a local change is accepted if:
    - improves the current state
    - makes the state worse,
      but such a state is accepted only with
      some probability dependent on "temperature"
  - „temperature" is continuously decreased so the probability of accepting a worsening step is also decreasing – a **cooling scheme** is used to define how the temperature decreases

Algorithm SA



```
procedure SA(InitT, MinT, MaxMoves)
    s ← random assignment of variables
    best ← s
    T ← InitT
    while MinT<T do
        num_errors ← 0
        while num_errors<MaxMoves do
            next ← a random local change of s
            if eval(next) < eval(s) then
                s ← next
                if eval(s) < eval(best) then best ← s
            else
                p ← random number in [0,1)
                if p < $e^{(eval(s)-eval(next))/T}$ then
                    s ← next
                else
                    num_errors ← num_errors+1
        end while
        T ← 0.8 x T
    end while
    return best
end SA
```

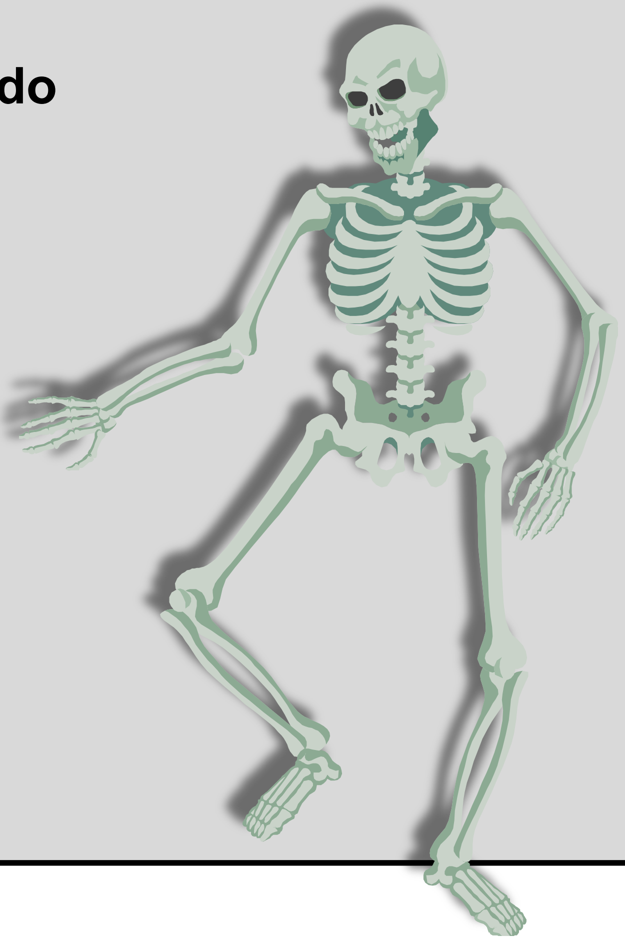
Metropolis heuristic

Localizer

The local search algorithms have a similar structure that can be encoded in the common skeleton. This skeleton is filled by procedures implementing a particular technique.

Local Search Skeleton

```
procedure local-search(Max_Tries,Max_Moves)
    s ← random assignment of variables
    for i:=1 to Max_Tries while Gcondition do
        for j:=1 to Max_Moves while Lcondition do
            if eval(s)=0 then
                return s
            end if
            select n in neighbourhood(s)
            if acceptable(n) then
                s ← n
            end if
        end for
        s ← restartState(s)
    end for
    return best s
end local-search
```



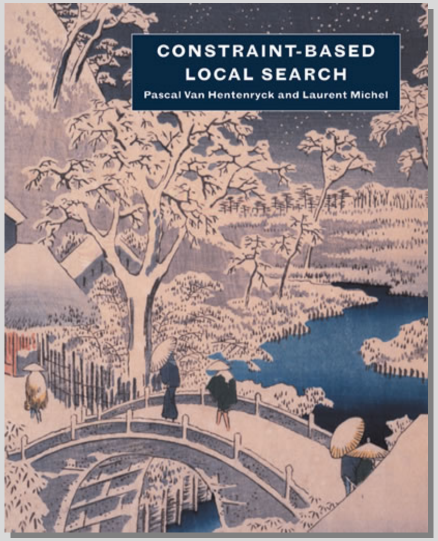
Local Search - Summary

Local search techniques start from some state and by moving to neighbouring states they try to reach a goal state.

Each algorithm is specified by:

- state neighbourhood and allowed states in the neighbourhood
- heuristic to select the next state from the neighbourhood (**intensification**)
- meta-heuristic to escape local optima (**diversification**)

`www.comet-online.org`



Localizer was the base of the **Comet** system (MaxOS X, Linux, Win), that allows description of local search algorithms in a declarative way.

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz